Toward a scalable refinement strategy for multilevel graph repartitioning

Sébastien Fourestier, François Pellegrini

Toward a scalable refinement strategy for multilevel graph repartitioning

Sébastien Fourestier, François Pellegrini

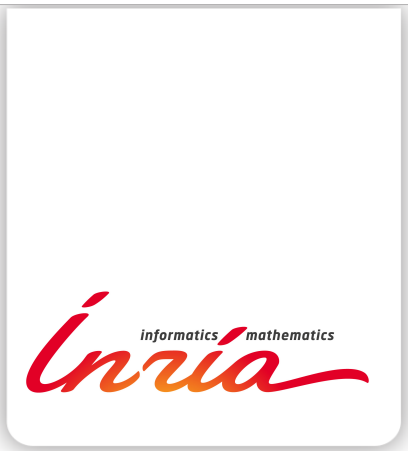
RESEARCH REPORT

N° 8246

February 2013

Project-Team Bacchus

ISSN 0249-6399 ISRN INRIA/RR--8246--FR+ENG



Toward a scalable refinement strategy for multilevel graph repartitioning

Sébastien Fourestier, François Pellegrini*

Project-Team Bacchus

Research Report n° 8246 — February 2013 — 22 pages

Abstract: Dynamic load balancing is a mandatory feature for parallel software whose workload evolves with time, such as solvers implementing adaptive mesh refinement. In such solvers, problem space is most often represented as an unstructured mesh, and graph partitioning is used to distribute data and their associated computations across processes.
The purpose of this paper is to study the sequential version of a set of graph repartitioning methods and to propose a scalable strategy for parallel graph repartitioning. As the repartitioning methods have been adapted from existing algorithms that are used for parallel partitioning, we will also be able to discuss their parallel behaviors. These methods can be combined in several ways, leading to either multilevel diffusion-based or biased scratch-remap frameworks. The proposed repartitioning framework uses a global diffusion-based algorithm for partition refinement, which may prove a good replacement for Fiduccia-Mattheyses type algorithms as it is inherently parallel. This algorithm yields best results when used in a multilevel framework.
To validate our approach, we compare our sequential graph repartitioning implementation within the Scotch software to the ParMeTiS graph repartitioning tool.

Key-words: Dynamic load balancing, graph repartitioning, diffusion, multilevel framework.

* {fouresti|pelegrin}@labri.fr

RESEARCH CENTRE BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

Vers une stratégie scalable pour un repartitionement avec schéma multi-niveaux

Résumé : L'équilibrage dynamique de la charge est une fonctionnalité indispensable aux applications parallèles dont la quantité de calcul évolue en fonction du temps, tels les solveurs utilisant du raffinement de maillage. Dans le cas de ces solveurs, l'espace du problème est le plus souvent représenté par un maillage non structuré et l'on utilise le partitionnement de graphes afin de distribuer sur les processeurs les données et les calculs associés.

L'objectif de cet article est d'étudier la version séquentielle d'un ensemble de méthodes de repartitionnement de graphes et de proposer une stratégie scalable pour le repartitionnement parallèle de graphes. Étant donné que ces méthodes de repartitionnement ont été adaptées à partir d'algorithmes existants utilisés pour le partitionnement parallèle, nous aborderons aussi leur comportement parallèle. Ces méthodes peuvent être combinées de plusieurs manières, aboutissant à des plateformes de repartitionnement basées soit sur une diffusion multi-niveaux, soit sur une méthode de type *scratch-remap* biaisée. La plateforme de repartitionnement que nous proposons utilise un algorithme de diffusion global pour le raffinement de partitions. Celui-ci, du fait qu'il est intrinséquement parallèle, pourrait constituer un bon remplaçant des algorithmes de type Fiduccia-Mattheyses. Cet algorithme donne de meilleurs résultats lorsqu'il est utilisé au sein d'un schéma multi-niveaux.

Afin de valider notre approche, nous comparons notre implémentation d'algorithmes de repartitionnement séquentiel de graphes, réalisée au sein du logiciel Scotch, au logiciel de repartitionnement de graphes ParMeTiS.

Mots-clés : Équilibrage dynamique de la charge, repartitionnement de graphes, diffusion, schéma multi-niveaux.

1 Introduction

Because of the huge amount of computation and data that they involve, large scale scientific simulations can only be performed by means of parallel processing. Machines that perform large scale simulations now comprise several tens of thousands of processing elements (which we will generically call "processors"). They are built using the distributed memory paradigm, because shared memory access would result in too much congestion. In order to use efficiently these machines, one has both to spread evenly computation load across processors, and to minimize communication induced by the exchange of information between neighboring processors.

When processes coexist for the entire duration of the parallel program, the load balancing problem can be modeled as a constrained graph partitioning problem. Distributing computations across p processors amounts to partitioning into p parts an undirected weighted graph, called *process graph*, whose vertices represent the computations to be performed, and whose edges represent computation inter-dependencies. The objective function to be optimized by the partition consists in minimizing cut size (that is, the sum of the weights of edges whose ends belong to two different parts) while evenly balancing the weights of the parts (that is, the sum of the weights of all the vertices that are assigned to each of them). Once the partition is computed, a process is associated with every part. Each process will perform all computations assigned to its part, and will hold all of the associated data.

However, in many application domains (for instance, in the case of mesh adaptation), the amount of computation associated with vertices may evolve with time, or vertices may be added to, or removed from the graph. In this case, the initially balanced distribution may become imbalanced, so that a new distribution must be computed so as to preserve processor efficiency. The purpose of this new partition is to obtain a balanced distribution anew, that will still minimize cut size, but that will also minimize the amount of data that has to be redistributed across processors. In order to be applicable to graphs of large sizes, the graph repartitioning process must be parallel, because it has to be performed on the fly, at fixed times or when the observed imbalance becomes too high.

The purpose of this paper is to study ways to combine and extend existing partitioning algorithms so as to devise a scalable strategy for parallel graph repartitioning. The next section is devoted to a state of the art of the methods most commonly used for graph repartitioning. Section 3 describes how we extended several graph partitioning algorithms to the repartitioning problem, with scalability in mind. We present in Section 4 the experimental results that we obtain with the sequential versions of these algorithms, with respect to those produced by the PARMETIS software [12] (the latter being used because the sequential METIS software does not possess repartitioning capabilities). According to our findings, we conclude by proposing a promising scalable strategy for parallel graph repartitioning.

2 Related work

The graph repartitioning problem has already been well studied. Like the plain graph partitioning problem, it is NP-hard [9]. Consequently, for problem sizes that interest us (that is, graphs above a billion vertices), it can only be addressed by means of parallel processing. Most of the numerous algorithms proposed in the literature (see [3] and its included references) can be categorized in two main classes.

Scratch-remap methods [15, 20] decouple the load balancing problem from the problem of determining the vertices to migrate. They partition from scratch the new (modified) graph, after which they associate the new parts to the processors so that the number of migrated vertices is minimal. Diffusion methods [7, 11, 19, 22, 23, 24] iteratively update an existing partition, by

migrating vertices that belong to the borders of the most heavily loaded parts to their neighboring parts, little by little, until load imbalance decreases below a prescribed threshold.

Scratch-remap methods favor the optimization of metrics that are taken into account during plain partitioning, that is, mostly load imbalance and cut size, to the detriment of the minimization of the number of vertices to migrate. Diffusion methods, because of their intrinsically local behavior, yield good results when the new graph is structurally close to the old one, but can bring a globally non-optimal solution when topological and/or vertex weight modifications are important. Moreover, these methods are expensive and parallelize poorly, whether migration is performed iteratively [20], or else computed by means of a linear optimization solver [14].

A trade-off between these two classes of methods consists in applying regular graph partitioning algorithms to augmented graphs which integrate, as additional vertices and edges, information regarding vertex migration [2, 4, 21]. To do so, as many *anchor* vertices are added to the new graph as there were parts in the initial partition of the old graph. The weight of these anchor vertices is assumed to be zero, so that they do not interfere with the load balancing process. Every vertex that belonged to one of the old parts is connected to the relevant anchor vertex by a fictitious edge. Hence, migrating a vertex to another part increases edge cut by the weight of this fictitious edge. It allows one to merge the vertex migration minimization objective function into the edge cut minimization function. The weights of fictitious edges define the migration cost of the vertices that they link, with respect to the communication cost represented by the weights of regular edges.

These *skewed* graph partitioning techniques, already used in the context of the static mapping of process graphs onto architecture graphs [10, 16], are both flexible and elegant. With only slight modifications to existing graph partitioning algorithms, they allow one to optimize concurrently load balance, cut size, and the amount of data to be redistributed across processors. Our work is based on this approach.

3 Adaptation of existing algorithms to repartitioning

3.1 A multilevel framework for computing k-way repartitions

Like the majority of today's graph partitioning tools, the Scotch [18] software that we used as a development testbed is based on a multilevel framework [1].

In this framework, graphs are repeatedly coarsened, by matching neighboring vertices, until the resulting coarsest graph is considered small enough. An initial mapping is computed on this coarsest graph, using methods that might have been too expensive to be used on the original graph. Then, this coarsest mapping is prolonged back, from coarser to finer graphs, to yield a mapping of the original graph. In order to ensure that the granularity of the produced solution is that of the original graph and not that of the coarsest graph, the prolonged mappings are refined at every level by means of local optimization algorithms. The combination of high quality initial mapping methods and of fast local optimization algorithms allows one to obtain good partitions in reasonable time. The computation of a partition through the multilevel method is commonly called a *V-cycle*.

Until now, Scotch used to compute k-partitions by means of a recursive bipartitioning algorithm, in which the obtainment of each bipartition required a complete multilevel v-cycle. Algorithms for locally optimizing bipartitions are easy to implement and fast, because every vertex of a bipartition can only move to the other part, resulting in simple data structures. However, the cost of repeating the v-cycle process quickly dominates run time when the number of parts increases.

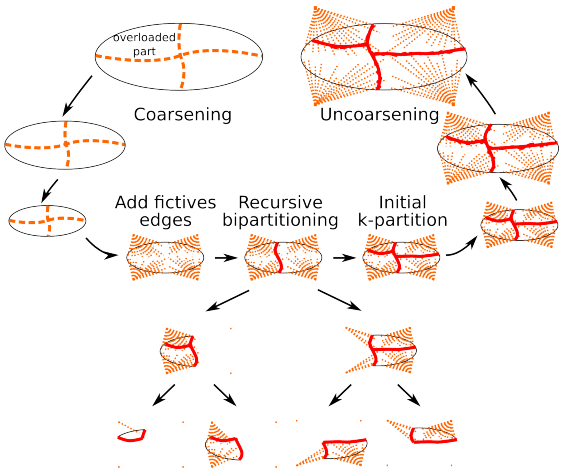


Figure 1: Adaptation of the multilevel framework to the repartitioning problem.

Since our repartitioning algorithms are expected to operate on very large graphs, we are bound to use a direct k-way partitioning framework, in which the v-cycle is only run once on the original graph. Consequently, one of the preliminaries of our work on repartitioning has been to add direct k-way partitioning capabilities to Scotch, like other tools already have.

We will now discuss four issues related to the adaptation of our direct k-way multilevel framework to the repartitioning problem: the handling of migration costs, the coarsening phase, the initial partitioning phase, and the uncoarsening phase.

3.1.1 Handling of migration costs

As introduced in the previous section, and as depicted in Figure 1, repartitioning capabilities have been implemented in Scotch by adding fictitious edges to the graphs to repartition. These edges induce an increase in edge cut size every time a vertex is migrated out of its original part.

In the repartitioning API (*Application Programmer Interface*) that we implemented, the user has to provide both the old partition and the individual migration cost associated with every fictitious edge. Hence, migration costs may not necessarily be equivalent to vertex weights.

In Scotch, as in many other graph partitioning tools, vertex and edge weights are stored as integers. This historical design choice was made because integer arithmetic is generally faster than floating point arithmetic, and most of all because it is not subject to rounding problems. This is critical for iterative algorithms, for which a sequence of vertex moves should always yield the same partition state, irrespective of the order in which individual vertex moves are performed. Else, infinite loops might occur.

However, migration costs may not always be multiples of edge weights. They may even be smaller than the unit edge weight. In order to accommodate for such cases, in our API, migration costs are provided by means of two parameters: an array of integer costs, of a size equal to the number of vertices, and a single floating-point value. This value represents the factor by which integer migration costs should be multiplied before being accounted for in the cut cost metric. This floating point value is internally approximated as a fraction, and integer edge weights and integer migration costs are scaled according to the denominator and the numerator of this fraction, respectively. Thanks to this scaling, Scotch can still rely on integer arithmetic for its computations.

While fictitious edges are an interesting model, adding them to large graphs may pose a serious efficiency problem in terms of memory occupation and run time. This is why, in as many of our algorithms as possible, we have tried to simulate their presence rather than effectively create them, as we will see below.

3.1.2 Coarsening

In order to compute the initial repartition of the coarsest graph, the state of the old partition must be propagated to it. This can be done in two different ways.

A first approach is to add fictitious edges to the original graph at the beginning of the coarsening phase. Graph size must be increased by as many anchor vertices as there were parts in the old partition, and by as many fictitious edges as there are vertices in the new graph that belonged to the old graph. This would make the coarsening phase slower, and would oblige the coarsening algorithm to process anchor vertices in a special way, so that they are not mated to regular vertices.

A second approach is to create fictitious edges only after the coarsening phase, when computing the initial repartition on the coarsest graph. Since fictitious edges connect a regular vertex to its old part, every coarsest vertex must belong to only one part. Consequently, vertex mating can only take place between vertices belonging to the same part.

We chose to implement this second solution, as it only required the addition, to the mate selection routine, of a test that checks that a vertex and its prospective mate do not belong to different old parts.

3.1.3 Initial partitioning

An initial k-way partition of the coarsest graph can be computed easily by using the existing recursive bipartitioning method. Yet, this method has to be adapted to compute an initial k-way repartition. Like for coarsening, two approaches can be followed.

A first approach is to add fictitious edges to the coarsest graph. The memory footprint is very limited, but it also requires to manage anchor vertices in all the graph bipartitioning algorithms, so that these vertices can never be moved out of the part that they represent. Adding more tests to every algorithm unduly makes them more complex.

A second approach is to take advantage of an existing feature of the recursive bipartitioning method of Scotch, which already implements biased graph bipartitioning algorithms. Indeed, Scotch not only does graph partitioning, but also computes static mappings onto graphs that represent machine processor topologies [16]. When deciding whether a vertex must be kept into its current part or moved to the other part, bipartitioning algorithms account for the local cut, but also for the fact that changing the vertex part may move it closer (resp. farther) to vertices that have already been mapped onto distant processors. To handle this, an *external gain* value is associated with each vertex, which represents the bias imposed by the outside environment. By adding migration costs to the external gains of every vertex, repartitioning can be managed at almost no cost.

3.1.4 Uncoarsening

During the uncoarsening phase, the k-partition computed at a coarser level is prolonged to the finer graph. Every pair of mated vertices is assigned to the part to which the coarse vertex that represented it in the coarser graph is assigned. Once this is done, local optimization algorithms are used to smooth the borders of the prolonged partition.

Two methods are used for this purpose, that will be presented in detail below.

3.2 Refinement heuristics

We will present in this section two refinement algorithms that are used during the uncoarsening phase of our multilevel framework: a k-way version of the Fiduccia-Mattheyses heuristic [8], and a diffusion method.

Bipartitioning versions of these algorithms already existed in the previous version of Scotch. Our work has consisted in extending them to the k-way domain and adapting them to the repartitioning problem.

3.2.1 K-way Fiduccia-Mattheyses heuristic

This method is classically used in many partitioning software, such as ParMeTiS. This method iteratively moves vertices belonging to the borders of the parts, trying to minimize the edge cut while maintaining load balance within a user-specified tolerance.

This method can be applied almost straightforwardly to a graph to which fictitious edges have been added. The only modification amounts to adding specific tests so that anchor vertices will never be moved. These tests amount to not letting anchor vertices be placed into the data structures that contain boundary vertices suitable for moving. This code also serves for fixed vertices; indeed, anchor vertices are a special kind of fixed vertices.

Although the use of fictitious edges is straightforward in this case, we have decided not to rely on them in our implementation. Rather, we model their impact as an extra migration cost when computing the gain associated with the move of a vertex to a part to which it does not belong. Doing so spares the time and memory required to build the augmented graph with fictitious edges. It will also allow us to overcome a problem that arises when the old partition becomes too imbalanced, as we will see below.

All Fiducia-Mattheyses-like algorithms make a heavy use of some flavor of bucket data structure. The purpose of this structure is to sort all possible vertex moves per descending edge cut gain. To find the next potential vertex to move, the algorithm extracts vertices from this structure one by one, and checks whether the move would violate the load imbalance constraint. If it would, the vertex is put aside and the next best vertex is extracted from the structure. Once a possible vertex has been found, or even if none are found, all rejected vertices are put back into the structure.

While such an implementation works well for graph partitioning, it may lead to dramatic slow-downs when applied to graph repartitioning, especially when migration costs are high and when vertex weights evolved much between the old and the new graphs. Because migration costs are high, the algorithms will try to move as few vertices as possible. Because vertex weights changes, it has to move enough vertices so as to produce a balanced new partition from an old partition that has become unbalanced. The best partitions will be the ones that are as close as possible to the imbalance threshold.

Consequently, all the vertices with highest gains, from the edge cut minimization perspective, are the ones that cannot be accepted because they would violate the load imbalance constraint. The bucket structure is heavily solicited but mostly yields invalid vertices, resulting in poor performance.

We tackled this problem in the following way: when a vertex with a migration gain is put aside, it will be put back in the bucket structure according to its edge cut gain only, without considering again its migration gain. Hence, it will not be likely to be considered again in the next searches to come, while never being placed below vertices without migration gains of worse quality in terms of cut cost minimization. It will regain its migration gain when one of its neighbors is moved, or at the next iteration of the outer, global loop of the Fiduccia-Mattheyses

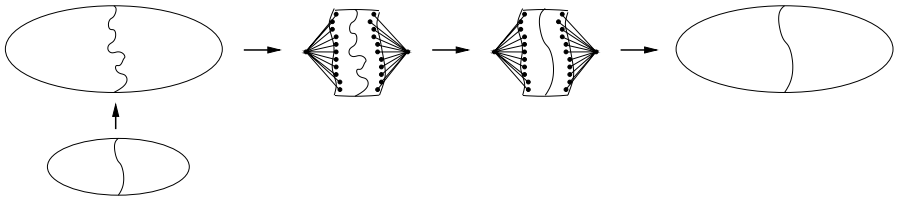


Figure 2: Banded multilevel refinement method, for a case with two parts. A band graph of small width is created around the prolonged frontier. All the remaining vertices in each part are coarsened into a single anchor vertex. After a local optimization algorithm is applied, the refined band frontier is prolonged to the full graph. This enables the multilevel uncoarsening process to go on at the upper levels.

algorithm. As we will see in the next section, this optimisation does not significantly degrade partition quality, while preserving run time for high migration costs.

The Fiduccia-Mattheyses method computes good partitions while preserving load balance. Yet, it has two drawbacks. Because it performs only local optimization, it can yield frontiers made of segments of locally optimal cut, but whose juxtaposition is not globally optimal. Also, it does not parallelize well, because it is inherently iterative: vertices are moved one by one, because every move of a vertex requires to recompute the gains of its neighboring vertices.

It is to alleviate these two problems that we also considered a global diffusive method.

3.2.2 Global diffusion-based heuristic on band graphs

This algorithm has been pictorially called "*the Jug of the Danaides*" [17], because of its analogy with its mythological counterpart. It models the graph as a set of leaking barrels, representing its vertices, linked by pipes, representing the edges. Each of the parts receives a given amount of liquid of a different color per unit of time. Liquids can flow freely through the pipes according to their section, and two different liquids vanish in equal quantities when they mix. Only the dominant liquid will remain in each barrel, leading to the definition of areas that minimize their interface with respect to their inside, as soap bubbles do.

Since this algorithm is targeted towards locally optimizing prolonged partitions, only the immediate vicinity of current frontiers is relevant to us. Consequently, we restrict its use to a *band graph* [5], defined by the restriction of the prolonged graph to the set of vertices that are at distance at most d from the current frontiers (by default, this distance is set to 3). All the removed vertices of each part are coarsened into a single anchor vertex representing this part. These anchor vertices are linked to the vertices of same part that belong to the last layer of the band graph, as illustrated in Figure 2. It is from these anchor vertices that the liquids will be injected in the network of barrels.

Subfigure 4(a) illustrates how this algorithm enforces the balance constraint. As vertex 8 has a higher load, it leaks more liquid per time step than others (the amount of leaked liquid can be seen as the rental cost for occupying the barrel). Thus, even if it gets a lot of liquid from vertex 10, it will forward only a small fraction of it to vertex 5.

Just like the Fiduccia-Mattheyses algorithm, the adaptation of the *Jug of the Danaides* to graph repartitioning requires some special treatment, especially for high migration costs.

In this case, if the plain algorithm were directly applied to the graph to which fictitious edges have been added, *migration edges* would be of a higher diameter than *regular edges*. Because the

Table 1: Description of the test graphs. The vertex and edge cardinalities, $|V|$ and $|E|$, are given in thousands.

Graph	**Description**	**Size** ($\times 10^3$)		**Average degree**
		$\|V\|$	$\|E\|$	
`10millions`	3D electromagnetics, CEA	10423	78649	15.09
`af_shell10`	structural problem	1508	25582	33.93
`audikw_1`	structural problem	943	38354	81.28
`cage15`	DNA electrophoresis	5154	47022	18.24
`conesphere1m`	3D electromagnetics, CEA	1055	8023	15.21
`coupole8000`	3D structural mechanics, CEA	1768	41656	47.12
`dielFilterV3real`	electromagnetics problem	1102	44101	79.98
`ecology1`	2D/3D problem	1000	1998	4.00
`ldoor`	structural problem	952	22785	47.86
`thermal2`	thermal problem	1228	3676	5.99

dominant connections would link vertices belonging to the same old parts, the partition would not change, and remain unbalanced in the context of the new graph.

To successfully extend this algorithm to the repartitioning case, we inspired ourselves from the *influence model*. In this diffusive model, explored by Wan *et al.* [25], vertices impact their neighbors by diffusing them information on their current state, but this mutual influence is reduced when neighbors do not belong to the same part.

Figure 3 shows the sketch of our extended algorithm. On Figure 4(b), our algorithm would move vertex 5 from Part 3 to Part 1. If a vertex has neighbors that are of its old part and others that are in other parts, it can redirect some liquid to help in decreasing migration. In practice, an amount of the *standard liquid* that cross these vertices, and which is proportional to their migration cost, is converted into *migration liquid*. This migration liquid is then redirected so as to favor the coming back of the vertex to its old part. The 2 gives some red liquid to vertex 5 to help it come back to the first part while vertex 5 does not send green liquid to vertex 2 to reach a higher probability to come back to its old part. Since it will not help vertex 5 to come back to its old partition, vertices 7 and 8 will not send their migration liquid to it.

The benefit of this method is that it gives good cut while taking load balance constraint in account. Moreover, it is global, scalable and easily parallelizable. Yet, it has three drawbacks: it is more expensive than the Fiduccia-Mattheyses heuristic; the load balance tolerance cannot be chosen; and we cannot use it on graphs from which the k-way band graph cannot be extracted (usually when there are lots of parts and the graph has a high degree).

4 Experimental results

The results presented in this paper have been computed on the nodes of the PLAFRIM cluster, each blade comprising two quad-core Nehalem Intel® Xeon® X5550 processors running at 2.66 GHz and 24 Gb of main memory.

4.1 Protocol

To perform our experiments, we considered graphs which have unit weights and come from various domains. They are described in Table 1.

We considered four metrics:

- *Cut*: let $G = (V, E)$ be a graph, and let w_e be the load of an edge $e \in E$. We call S the separator, that is the set of edges that cross two parts. Our *cut* metric is a ratio defined as $\frac{\sum_{s \in S} w_s}{\sum_{e \in E} w_e}$.
- *Imbalance*: for any partition, let $|V_p|$ be the load sum of vertices in part $p \in P$. Our *imbalance* metric is a delta ratio defined as $\frac{\sum_{p \in P} \left| |V_p| - \frac{|V|}{k} \right|}{|V|}$.
- *Migration*: for an initial partition and a repartitioning, let V_m be the set of vertices that belong, in the repartitioning, to a part that differs from their initial part. Our *migration* metric is a ratio defined as $\frac{|V_m|}{|V|}$.
- *Time*: the execution time, in seconds. Scotch is executed on one processor and ParMeTiS on two processors. For the sake of clarity and even if it takes advantage from its parallel execution, we have chosen to keep ParMeTiS run time as is, without any kind of normalization per processor.

Our protocol for testing repartitioning strategies is as follows.

1. We compute an *initial partitioning* of 128 parts with the default strategy of Scotch 6.0 (`sq`) and a balance constraint of 0.05[1].
2. We compute a *modified graph* by increasing by one the weights of the vertices that are in the first 32 parts of the *initial partitioning*. This brings an imbalance ratio of about 0.16.
3. We use various strategies to compute a repartitioning (that keeps 128 parts) of the *initial partitioning* on the *modified graph* with respect to a balance constraint of 0.05. For all graphs and all strategies, this step is done 100 times for several migration costs comprised in the range $[0.5, 50]$[2].

The strategies that we experimented with are the following.

- `rb`: The initial partition of the k-way multilevel framework is computed thanks to *Recursive Bipartitioning*, and is uncoarsened without refinement.
- `rbf`: The initial partition of the k-way multilevel framework is computed thanks to *Recursive bipartitioning*. During the uncoarsening phase, we use the *Basic Fiduccia-Mattheyses* refinement.
- `rf`: The initial partition of the k-way multilevel framework is computed thanks to *Recursive bipartitioning*. During the uncoarsening phase, we use the *Fiduccia-Mattheyses* refinement. This is the same strategy as `rbf` with the optimisation explained in subsection 3.2.1.
- `rd`: The initial partition of the k-way multilevel framework is computed thanks to *Recursive bipartitioning*. During the uncoarsening phase, we use the *Diffusion* refinement (cf. subsection 3.2.2).

[1] The desired imbalance ratio of 0.05 passed to Scotch turned into the filling of the ParMeTiS `ubvec` input array with 1.05 values.

[2] We increase the migration cost by steps of 0.01 when it is less than 1, and by steps of 1 afterward. The migration cost is translated into the `itr` parameter used in input of ParMeTiS, according to the following formula: $\texttt{itr} = \frac{1}{m}$

- `sq`: This is the default strategy of *Scotch* 6.0 using the `SCOTCH_graphRemap` routine[3] that privileges cut (*Quality*) over strict balance[4]. The initial partition of the k-way multilevel framework is computed thanks to recursive bipartitioning. During the uncoarsening phase, we use diffusion and Fiduccia-Mattheyses refinements.

- `pm`: This is the *ParMetis* 4.0.2 [13] strategy using the `ParMETIS_V3_AdaptiveRepart` routine.

4.2 K-way Fiduccia-Mattheyses heuristic behaviour

In Figure 5, we observe the behavior of Fiduccia-Mattheyses-like strategies on the `10millions` graph. `pm` yields a slightly better cut, while bringing a small extra to the balance constraint for migration costs higher than 1. `rbf` and `rf` are more sensitive to the variation of migration cost. `rf` is a bit more expensive than `pm` and, as explained in subsection 3.2.1, `rbf` has the most expensive execution time. As `rbf` explores more vertices, it is able to reach better balance.

In Figure 6, we observe, by strategy, the repartition of all runs. It confirm our first observations. `rbf` and `rf` yield almost the same cut and migration ratios. `rbf` can be very expensive but gives the best balance. `pm` gives a better cut but a worse balance, and is the fastest. The migration ratio of `pm` is often close to 40%. In the previous subsection, we explained how we converted the migration value to the input ParMeTiS parameter `itr`. In these experiments, we used an `itr` parameter that is included in the range $[0.02, 2.0]$. The ParMeTiS documentation claims that `itr` can be set in the range $[0.000001, 1000000.0]$. We tried it without observing major changes in the migration ratio.

As `rf` enforces the balance and gives the same cut and imbalance as `rbf` while being faster, we will focus on this implementation for the rest of our analysis.

4.3 Global diffusion-based heuristic behavior

In Figure 7, we observe the behavior on graph `10millions` of the diffusion strategy (`rd`), compared to `sq`, `pm` and the basic `rb` strategy. `rd` yields a cut that is less sensitive to the migration cost. It does not always enforce the balance constraint and it is more expensive than Fiduccia-Mattheyses-like strategies. Like `rf` and `sq`, `rd` is more sensitive to the migration cost than `pm`. `sq`, which combines `rd` and `rf`, brings a cut close to `pm`, which is better for high migration costs, and a better imbalance, but it is more expensive.

Because of the nature of the algorithm, the balance constraint cannot be configured for `rd`, which brings in practice an imbalance close to 0.05.

`rd` has the same behavior on the other graphs, save for `audikw_1`, `cage15` and `dielFilterV3real`. On these three high-degree graphs, our k-band graph algorithm does not complete because some parts do not contain three layers of vertices. As the k-ary band graph cannot be created, diffusion refinement is not performed and we get exactly the same behavior as `rb`.

For clarity of the analysis, we will now focus on the other graphs, to which we will collectively refer as *mesh-type* graphs.

Table 2: Execution time means for all runs on "mesh-type" graphs

Graphs	rb	rd	rf	sq	pm
10millions	10.603	111.150	45.645	118.542	17.056
af_shell10	1.548	13.149	3.453	14.564	2.040
conesphere1m	1.551	11.464	4.991	14.363	1.580
coupole8000	2.490	11.014	3.459	11.768	5.231
ecology1	0.533	2.095	7.290	2.650	0.700
ldoor	1.302	13.828	2.274	15.397	2.033
thermal2	0.778	3.289	1.906	4.126	1.108
Global	2.68655	23.713	9.860	37.370	4.250

4.4 Strategy analysis

4.4.1 Migration

In Figure 8, we observe that all Scotch strategies are more sensitive to the migration cost than the ParMeTiS strategy. `rd` always migrates more than other strategies. In comparison to `rb` and `rf`, `sq` migrates more for small migration costs, and less for high migration costs. All strategies exhibit an equivalent migration ratio for high migration costs.

4.4.2 Cut and imbalance

Figure 9 evidences that, as expected, `rb` yields the worst cut (0.0361). As the initial partition is balanced before the plain uncoarsening phase, it gives a good balance (0.0366). `pm` gives the best cut (0.0289) and `rf` gives the best balance (0.0340). `sq` gives a cut close to the one of `pm` (0.0297, with a ratio of 1.028), while bringing a better balance (0.0435, with a ratio of 0.852). On average, `pm` slightly exceeds the 0.05 balance constraint (0.001), while `rd` does so a little more (0.006). On average, `rd` gives the second best cut, close behind `pm` (0.0296 with a ratio of 1.025 and 53 % of its cut values that are less than the ones of `pm`). `rf` brings a respectable cut (0.0307).

4.4.3 Time

From table 2, we see that the execution time of `rb` is significantly smaller than others. Consequently, most of the execution time is consumed during the refinement phase. The Fiduccia-Mattheyses-like strategies are faster than the ones that comprise diffusion-like refinement. `pm` is close to them but faster than `rf` (its wall-clock time multiplied by two is still a bit smaller than that of `rf`). On average, running `rd` on one processor is 5.6 times more expensive than the execution of `pm` on two processors, and the execution of `sq` is 8.8 times more expensive.

4.5 Summary

To sum up, `pm` and `sq` are the two best strategies in terms of cut and imbalance. The Scotch strategies are more sensitive to the migration cost parameter. The Fiduccia-Mattheyses-like strategies provide a good cut while enforcing a parametrizable balance constraint, and are fast. `rd` gives a good cut, a load balance close to 0.05, and is more expensive.

Because `rd` is algorithmically more scalable than Fiduccia-Mattheyses-like strategies, a scalable strategy for parallel graph repartitioning could be the following:

[3]This strategy can be set by enabling the flag `SCOTCH_STRATQUALITY`.
[4]This strategy can be set by enabling the flag `SCOTCH_STRATBALANCE`.

1. Perform parallel coarsening so as to get a graph small enough to be stored on one processor.

2. On each processor, perform a sequential repartitioning of this small graph using the `sq` strategy.

3. During the uncoarsening phase, when the graph becomes too large to be stored on one processor, compute a k-way band graph and refine it sequentially using the diffusion algorithm followed by a Fiduccia-Mattheyses-like refinement.

4. When the band graph becomes too big to be stored on one processor, get the best computed repartitioning, compute a k-way parallel band graph and perform a parallel refinement thanks to the diffusion algorithm.

5 Conclusion and Future Work

This paper aims at proposing a strategy suitable for parallel graph repartitioning. To fulfill this goal, we performed an experimental analysis of several sequential repartitioning strategies. These strategies are based on methods that were specifically adapted to the graph repartitioning problem. This adaptation was made necessary because high migration costs and heavy imbalance put a high stress on existing partitioning methods, which prevents them from behaving properly on biased graphs. It is in this context that we presented our adaptation of the diffusion algorithm to the repartitioning problem.

Future work includes the completion of the coding of the parallel version of our repartitioning algorithms within the PT-Scotch software [6]. This will enable us to perform scalability studies of our proposed parallel graph repartitioning strategy. Another area of work is the improvement of our k-band graph algorithm so as to enable the use of diffusion refinement methods on high degree or non-mesh type graphs.

Acknowledgments

The results presented in this paper were obtained on the PLAFRIM test platform that was put in production thanks to the *action de développement INRIA PlaFRIM* with support from LaBRI, IMB and the following institutions: Conseil Régional d'Aquitaine, FeDER, Université de Bordeaux and CNRS (see: `https://plafrim.bordeaux.inria.fr/`).

The first author would like to thanks Emmanuel Jeannot and Louis-Claude Canon for introducing himself to the R-project.

References

[1] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117, 1994.

[2] Umit Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdag, Robert Heaphy, and Lee Ann Riesen. A repartitioning hypergraph model for dynamic load balancing. Sandia National Laboratories Tech. Report SAND2008-2304J, Sandia National Laboratories, Albuquerque, NM, 2008. Submitted to J. Par. Dist. Comp.

[3] Umit Catalyurek, Doruk Bozdag, Erik G. Boman, Karen D. Devine, Robert Heaphy, and Lee Ann Riesen. Hypergraph-based dynamic partitioning and load balancing. Sandia National Laboratories Tech. Report SAND2007-0043P, Sandia National Laboratories, Albuquerque, NM, 2007.

[4] Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdağ, Robert T. Heaphy, and Lee Ann Riesen. A repartitioning hypergraph model for dynamic load balancing. *J. Parallel Distrib. Comput.*, 69:711–724, August 2009.

[5] C. Chevalier and F. Pellegrini. Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. In *Proc. Euro-Par'06, Dresden*, volume 4128 of *LNCS*, pages 243–252, September 2006.

[6] C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34:318–331, 2008.

[7] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7:279–301, October 1989.

[8] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. 19th Design Automat. Conf.*, pages 175–181. IEEE, 1982.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractablility: A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[10] B. Hendrickson, R. Leland, and R. Van Driessche. Skewed graph partitioning. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*. IEEE, March 1997.

[11] Y. F. Hu, R. J. Blake, and D. R. Emerson. An optimal migration algorithm for dynamic load balancing. concurrency: Practice and experience, 1998.

[12] G. Karypis and V. Kumar. ParMetis *Parallel Graph Partitioning and Sparse Matrix Ordering Library*. University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN 55455, U.S.A., August 2003.

[13] MeTiS: Family of multilevel partitioning algorithms. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[14] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *J. Parallel Distrib. Comput.*, 69:750–761, September 2009.

[15] Leonid Oliker and Rupak Biswas. Plum: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52:150–177, 1998.

[16] F. Pellegrini. Static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proc. SHPCC'94*, pages 486–493. IEEE, May 1994.

[17] F. Pellegrini. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In *Proc. Euro-Par'07*, volume 4641 of *LNCS*, pages 191–200. Springer, August 2007.

[18] Scotch: Static mapping, graph partitioning, and sparse matrix block ordering package. `http://www.labri.fr/~pelegrin/scotch/`.

[19] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes, 1997.

[20] Kirk Schloegel, George Karypis, and Vipin Kumar. Wavefront diffusion and lmsr: Algorithms for dynamic repartitioning of adaptive meshes. *Trans. Parallel Distrib. Syst.*, 12:451–466, May 2001.

[21] C. Walshaw. Variable partition inertia: graph repartitioning and load-balancing for adaptive meshes. In S. Chandra M. Parashar and X. Li, editors, *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*. Wiley, New York, 2010. (Invited chapter).

[22] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. Technical report, Parallel Comput, 2000.

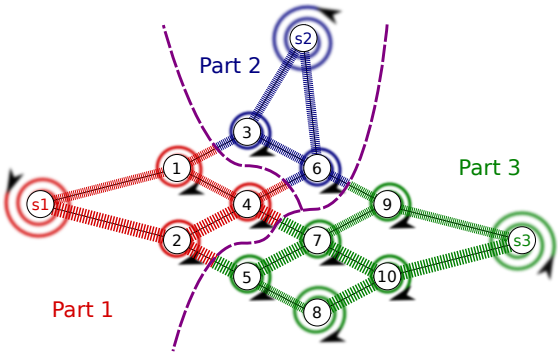
[23] C. Walshaw and M. Cross. Dynamic Mesh Partitioning and Load-Balancing for Parallel Computational Mechanics Codes. In B. H. V. Topping, editor, *Computational Mechanics Using High Performance Computing*, pages 79–94. Saxe-Coburg Publications, Stirling, 2002. (Invited chapter).

[24] C. Walshaw, M. Cross, and M. G. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108, 1997.

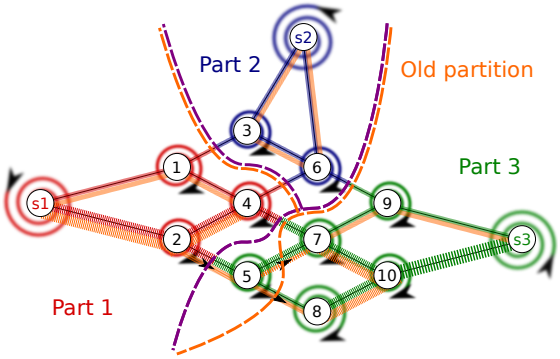
[25] Y. Wan, S. Roy, A. Saberi, and B. Lesieutre. A stochastic automaton-based algorithm for flexible and distributed network partitioning. In *Proc. Swarm Intelligence Symposium*, pages 273–280. IEEE, 2005.

initialize $\texttt{tab}_t$ and $\texttt{tab}_{t+1}$ arrays to 0; /* Initialization */
set `partval` of $\texttt{tab}_t$ array to initial partition;
while (number of passes to do) { /* Main loop */
  for (all k-parts p) { /* Refill sources vertices */
    $\sigma_{s_p} \leftarrow \sum_{e=(s_p,v')} \texttt{weight}[e]$; /* Sum weights of all adjacent edges */
    $\texttt{tab}_{t+1}[s_p].\texttt{diffval} \leftarrow \texttt{tab}_{t+1}[s_p].\texttt{diffval} + \frac{|V|}{k \times \sigma_{s_p}}$; /* Update diffusion value of source vertex */
  }
  for (all vertices v in graph) {
    reset contents of `liquid` array to 0;
    for (all edges $e = (v, v')$) { /* For all neighbours */
      $\texttt{diffval} \leftarrow \texttt{tab}_t[v'].\texttt{diffval}$; /* Get neighbour standard liquid contribution */
      if ($\texttt{part}[v'] = \texttt{orgpart}[v]$) /* Add its migration liquid contribution */
        $\texttt{diffval} \leftarrow \texttt{diffval} + \texttt{tab}_t[v'].\texttt{mdisval}$;
      else
        $\texttt{diffval} \leftarrow \texttt{diffval} + \texttt{tab}_t[v'].\texttt{mdidval}$;
      $\texttt{liquid}[\texttt{tab}_t[v'].\texttt{partval}] \leftarrow \texttt{liquid}[\texttt{tab}_t[v'].\texttt{partval}] + \texttt{diffval}$; /* Add its contribution to liquid array */
    }
    $p_{\max} \leftarrow \max(\texttt{liquid})$; /* Get part of most abundant liquid */
    $\texttt{tab}_{t+1}[v].\texttt{partval} \leftarrow p_{\max}$;
    $\texttt{diffval} \leftarrow \texttt{liquid}[p_{\max}]$; /* Get amount of most abundant liquid */
    $\texttt{diffval} \leftarrow \texttt{diffval} - \texttt{weight}[v]$; /* Leak liquid */
    $\sigma \leftarrow \sum_{e=(v,v')} \texttt{weight}[e]$; /* Sum weights of all edges adjacent to v */
    $\sigma_{\text{old}} \leftarrow \sum_{e=(v,v') \text{ and } \texttt{part}[v']=\texttt{orgpart}[v]} \texttt{weight}[e]$; /* Sum weights of edges going to oldpart(v) */
    $\texttt{migrval} \leftarrow 0$;
    if (($\sigma_{\text{old}} \neq 0$) and ($\sigma_{\text{old}} \neq \sigma$)) /* If redirection of liquid can reduce mig. */
      $\texttt{migrval} \leftarrow \texttt{migrcost}[v]$; /* Get vertex migration cost */
    if ($\texttt{migrval} > \texttt{diffval}$) { /* Remove it from the amount of standard liq. */
      $\texttt{migrval} \leftarrow \texttt{diffval}$;
      $\texttt{diffval} \leftarrow 0$;
    }
    else
      $\texttt{diffval} \leftarrow \texttt{diffval} - \texttt{migrval}$;
    $\texttt{tab}_{t+1}[v].\texttt{diffval} \leftarrow \frac{\texttt{diffval}}{\sigma}$; /* Set t+1 fraction of standard liquid */
    if ($\texttt{migrval} = 0$) { /* Handle migration liquid */
      $\texttt{tab}_{t+1}[v].\texttt{mdisval} \leftarrow 0$;
      $\texttt{tab}_{t+1}[v].\texttt{mdidval} \leftarrow 0$;
    }
    else {
      if ($\texttt{orgpart}[v] = \texttt{tab}_{t+1}[v].\texttt{partval}$) { /* If v will spread liquid of its old part */
        $\texttt{tab}_{t+1}[v].\texttt{mdisval} \leftarrow \frac{\texttt{migrval}}{\sigma_{\text{old}}}$; /* Mig. l. will go to vertices in the old part */
        $\texttt{tab}_{t+1}[v].\texttt{mdidval} \leftarrow 0$;
      }
      else {
        $\texttt{tab}_{t+1}[v].\texttt{mdisval} \leftarrow 0$;
        $\texttt{tab}_{t+1}[v].\texttt{mdidval} \leftarrow \frac{\texttt{migrval}}{\sigma - \sigma_{\text{old}}}$; /* Mig. l. will go to vertices in other parts */
      }
    }
  }
  swap $\texttt{tab}_t$ and $\texttt{tab}_{t+1}$ arrays;
}

Figure 3: Sketch of the jug-of-the-Danaides diffusion algorithm extended to repartitioning problem. For each step, the current and new contents of every vertex are stored in arrays $\texttt{tab}_t$ and $\texttt{tab}_{t+1}$, respectively.



(a) Partitioning



(b) Repartitioning

Figure 4: Extension of the diffusion algorithm to the repartitioning problem. Three *standard liquids* (red, blue and green) appears (the big spirals) from the source vertices `s1`, `s2` and `s3`. Standard vertices leak some liquid (the small spirals). The dashed lines represents the liquid flow going from one vertex to another. The line heaviness is proportional the quantity for liquid flowing across an edge. The orange liquid corresponds to the *migration liquid*, that is some *standard liquid* that is redirected so as to favor the migration constraints.

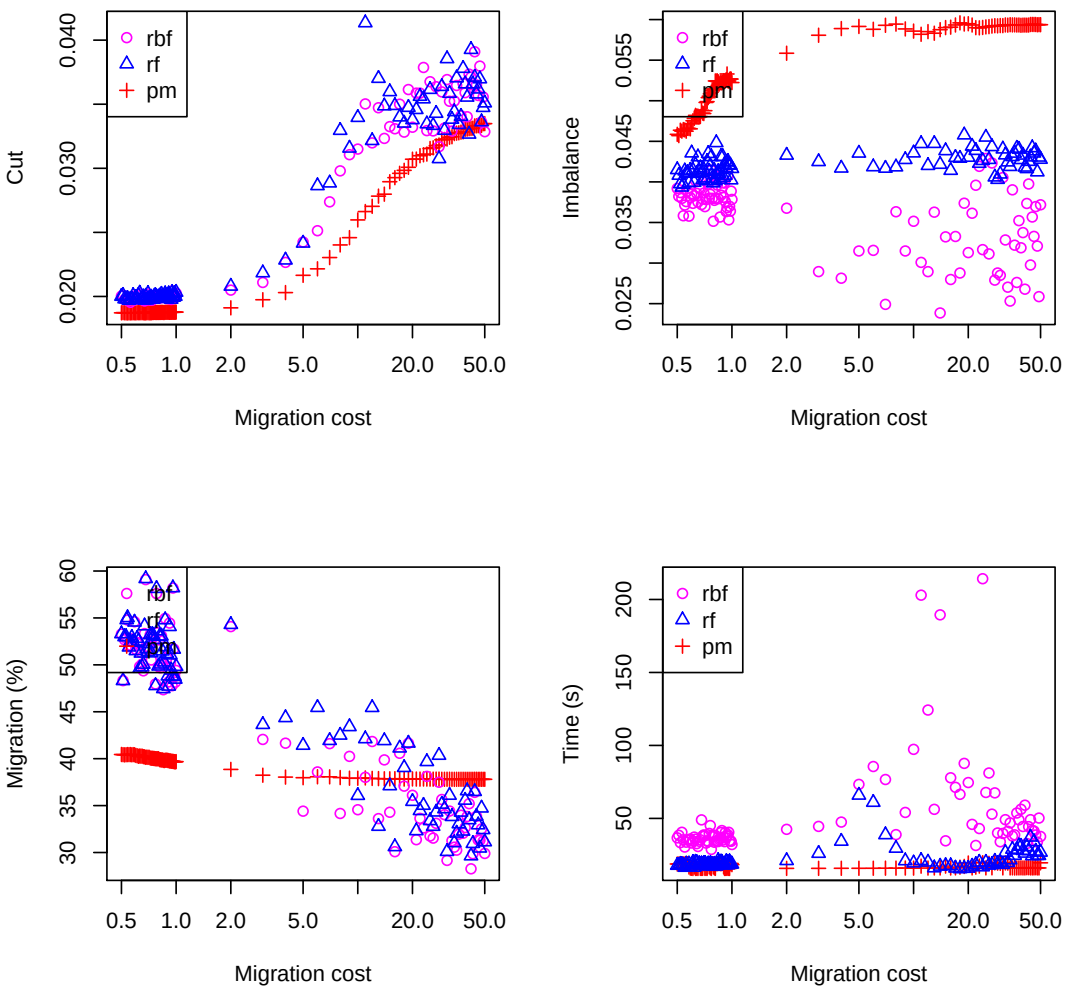


Figure 5: Behaviour, on the graph `10millions`, of the basic (`rbf`) and the optimized (`rf`) k-way Fiduccia-Mattheyses strategies implemented in Scotch, and the ParMeTiS strategy (`pm`).

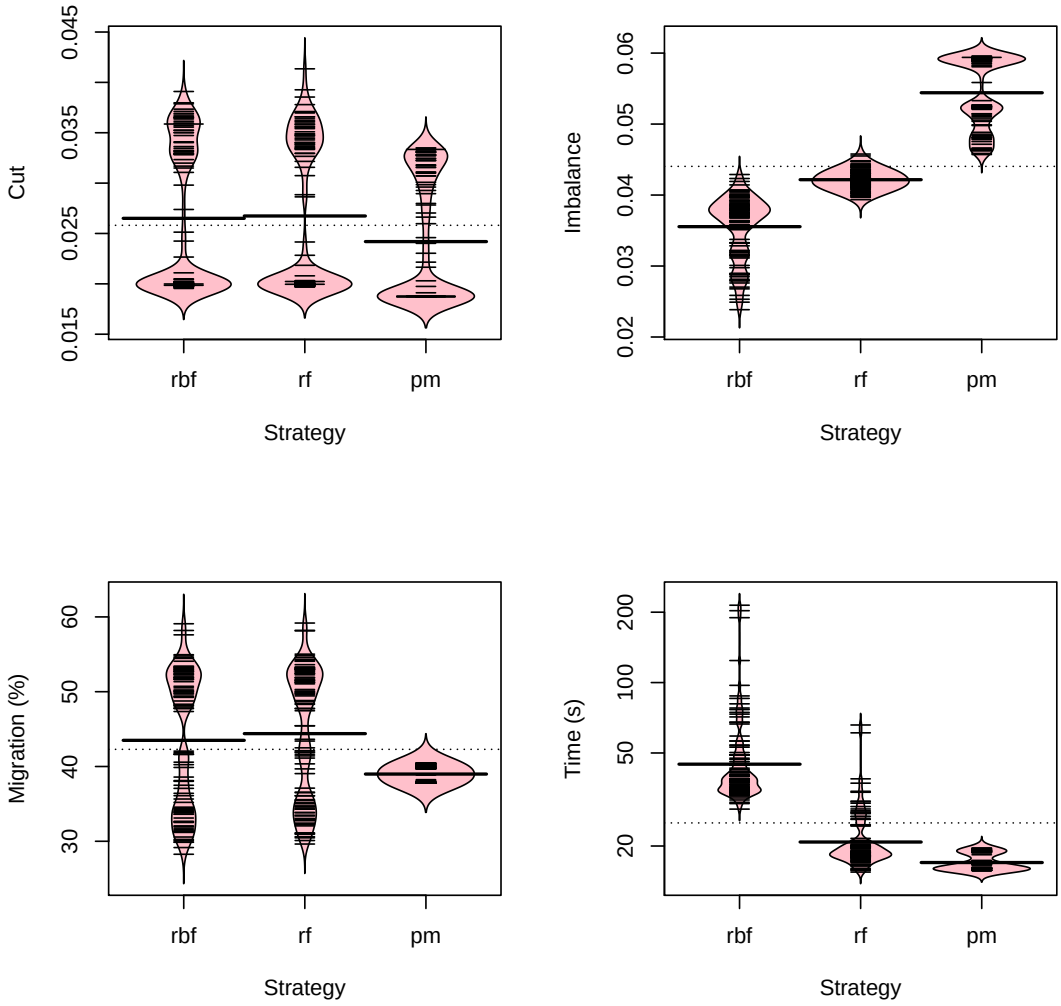


Figure 6: These bean plots give, for all runs with `rbf`, `rf` and `pm` strategies, the kernel density plot of the four considered indicators. The pink shape corresponds to a nonparametric density plot. The dashed horizontal line represents the global mean value for all strategies. The heavy, solid horizontal lines represents the mean for each strategy. The small horizontal lines corresponds to the values of each run within each *bean* of the beanplot.

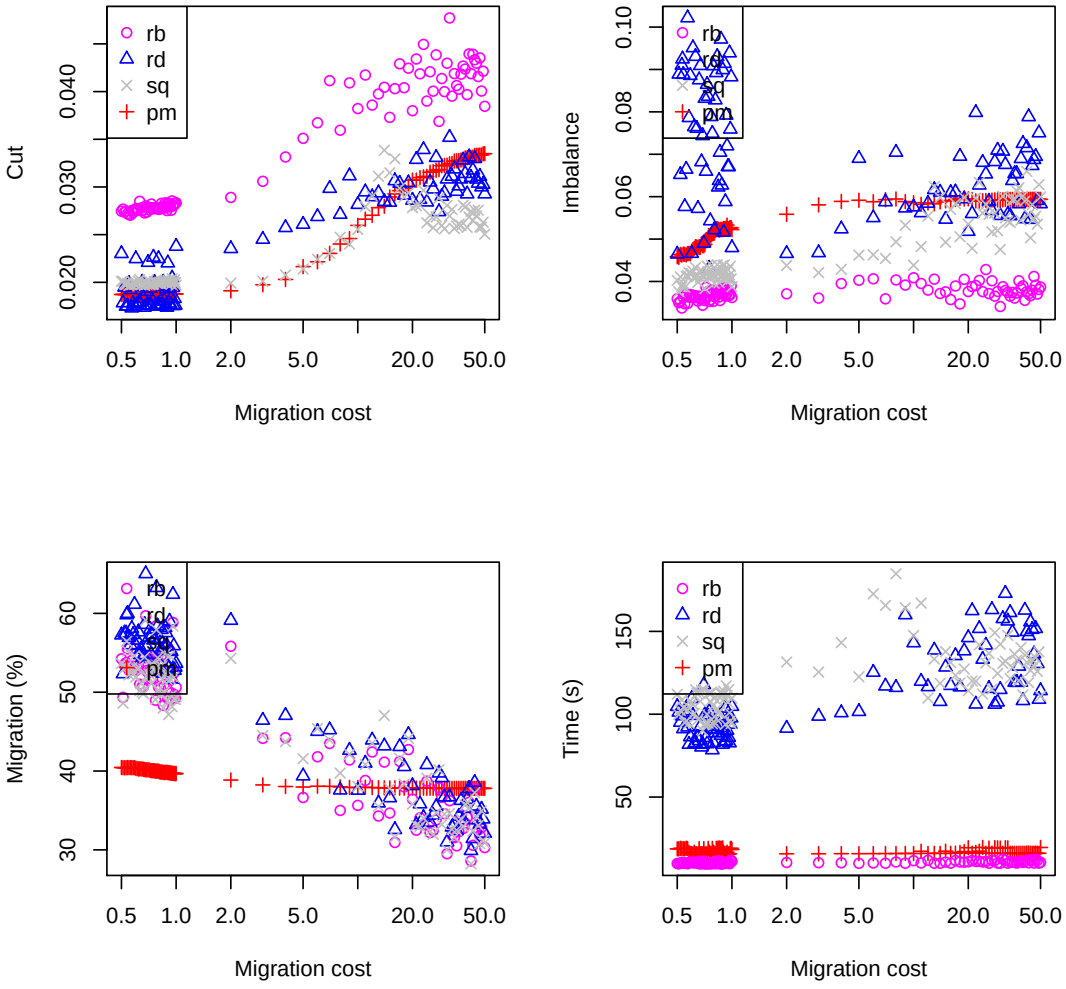


Figure 7: Behaviour, on the graph `10millions`, of the multilevel strategy without refinement (`rb`), the multilevel strategy with diffusion refinement (`rd`), the Scotch default strategy (`sq`) and the PaMeTiS strategy (`pm`).

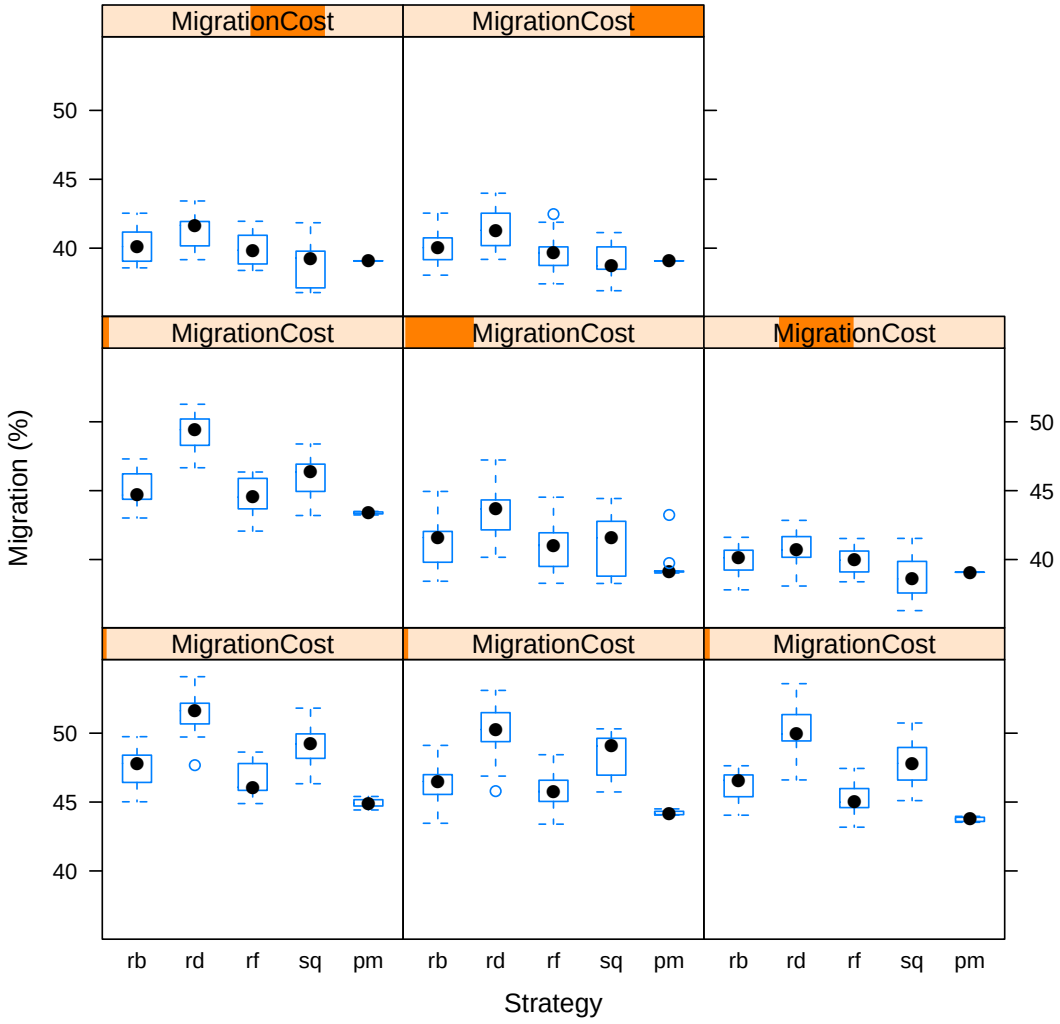


Figure 8: A trellis box-plot (that is a grid of several box-plots) showing, on "mesh-type" graphs, the migration behaviour of all the strategies for the following migration cost intervals: $[0.49, 0.61]$ (down left), $[0.62, 0.74]$ (down middle), $[0.75, 0.87]$ (down right), $[0.88, 0.99]$ (middle left), $[1, 12]$ (middle), $[13, 24]$ (middle right), $[25, 37]$ (up left) and $[38, 50]$ (up right).

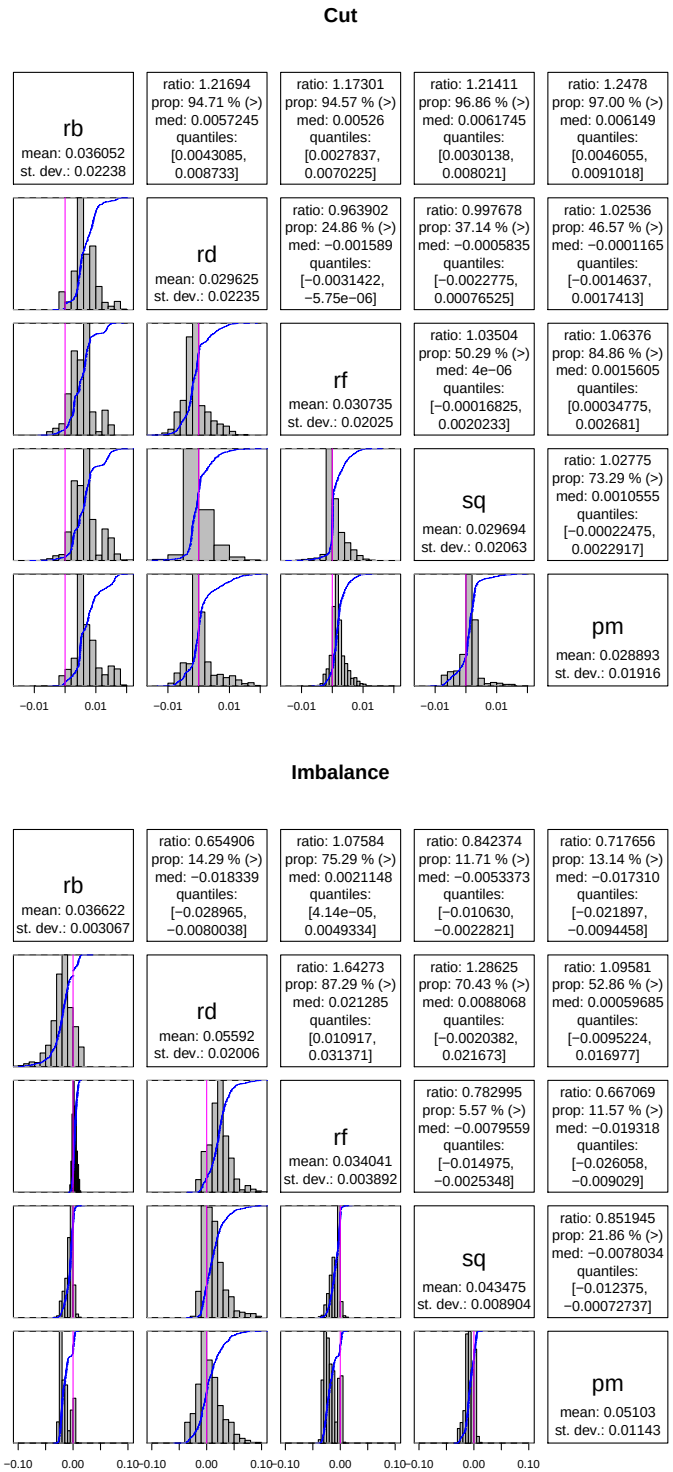
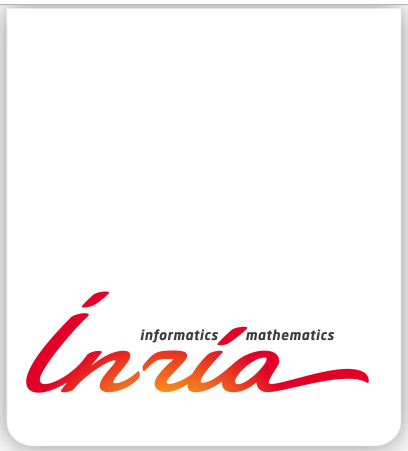


Figure 9: Comparison matrices showing, on "mesh-type" graphs, the strategies specificities in terms of cut and imbalance. On the diagonal, the strategy name, its mean and its standard deviation are given. On the lower part, the repartition of the difference between the runs of the above strategy and the ones of the right strategy is plotted. The empirical cumulative distribution function is plotted in blue. On the upper part, several metrics are given to ease the comparison between the left strategy (l) and the bottom one (b). They are defined as follows. `ratio` is equal to $\frac{\text{mean}(l)}{\text{mean}(b)}$. `prop` corresponds to the proportion of runs of the l strategy which are superior to the ones of the b strategy. `med` is the median of $l - b$. Eventally, the 25th and 75th quantiles of $l - b$ are given.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399